CSM2010 – Compiling Techniques
Course Assignment 2010 – 2011

Department of Computer Science. University of MALTA.

Sandro Spina / Gordon Mangion

This is the description for the assignment of unit CSA2010, Compiling Techniques. This assignment is worth 15% of the total mark for this unit. The assignment has to be carried out on an individual basis. Under no circumstances should code be shared among students. Please remember that plagiarism will not be tolerated; the final submission must be entirely your work.

Deliverables

You will submit your project source code, executables and a PDF file containing the project documentation/report on optical medium. Note that assignment binaries must run straight from CD/DVD-ROM without the need for any installation unless specified in the accompanying document, in which case a detailed installation guide must also be provided.

Description

In this assignment you are to develop an interactive parser and interpreter which will execute statements of a simple language called SFL (Simple Functional Language). Below is the definition in Extended Backus-Naur Form (EBNF) of the SFL language. The starting point of the grammar is the "*program*" non-terminal at the bottom of the grammar definition.

```
Digit             ::=   ["0"-"9"]
Letter            ::=   ["A"-"Z" "a"-"z"]
Type              ::=   "int" | "real" | "bool" | "char" | "string" | "unit"
BooleanLiteral    ::=   "true" | "false"
IntegerLiteral    ::=   Digit+
RealLiteral       ::=   Digit+ "." Digit* ( ("e"|"E") ("+"|"-")? Digit+ )?
CharLiteral       ::=   "'" (~["\"","\r","\n"]) "'"
StringLiteral     ::=   "\"" (~["\"","\r","\n"])* "\""
UnitLiteral       ::=   "#"
Literal           ::=   BooleanLiteral | IntegerLiteral | RealLiteral | CharLiteral | StringLiteral
                        | UnitLiteral
Identifier        ::=   ( "_" | Letter ) ("_" | Letter | Digit )*
MultiplicativeOp  ::=   "*" | "/" | "and"
AdditiveOp        ::=   "+" | "-" | "or"
RelationalOp      ::=   "<" | ">" | "==" | "!=" | "<=" | ">="
ActualParams      ::=   Expression ( "," Expression )*
```

```
FunctionCall       ::=   Identifier "(" ActualParams? ")"
TypeCast           ::=   "(" Type ")" Expression
SubExpression      ::=   "(" Expression ")"
Unary              ::=   ( "+" | "-" | "not" ) Expression
Factor             ::=   Literal | FunctionCall | Identifier | TypeCast | SubExpression | Unary
Term               ::=   Factor ( MultiplicativeOp Factor )*
SimpleExpression   ::=   Term ( AdditiveOp Term )*
Expression         ::=   SimpleExpression ( RelationalOp SimpleExpression )*
Assignment         ::=   Identifier "<-" Expression
VariableDecl       ::=   "let" Identifier ":" Type "=" Expression ( ";" | ( "in" Block )? )
FormalParam        ::=   Identifier ":" Type
FormalParams       ::=   FormalParam ( "," FormalParam )*
FunctionDecl       ::=   "function" Identifier "(" FormalParams? ")" ":" Type Block
IfStatement        ::=   "if" "(" Expression ")" Statement ("else" Statement)?
WhileStatement     ::=   "while" "(" Expression ")" Statement
HaltStatement      ::=   "halt" IntegerLiteral?
Statement          ::=   FunctionDecl
                         | Assignment ";"
                         | Expression ";"
                         | VariableDecl
                         | IfStatement
                         | WhileStatement
                         | HaltStatement ";"
                         | Block
Block              ::=   "{"  Statement* "}"
Program            ::=   Statement*
```

The language has Java-style comments, that is, //… for line comments and /*…*/ for block comments. The parser and interpreter are to be developed using JavaCC and Java. The type “unit”, which value is written as “#”, is used when an expression/statement does not have an actual value to return, equivalent to “void” in Java.

Task Breakdown

The assignment is broken down into four tasks. Below is a description of each task accompanied with the assigned mark.

Task 1 - Create Javacc grammar file

In this first task you are to create the Javacc grammar file for the SFL definition given above. You are free to modify the production rules as long as the changes are documented in the report and that the source language (SFL) remains unaltered. Should you prefer to use an alternative to the JJTree pre-processor in order to build the parse tree please go ahead.

[Marks: 25%]

Task 2 - Parse Tree Generation

You should enhance the parser developed in Task 1 to output a textual (or graphical if you prefer) representation of the generated parse tree.

```
E.g. Let X : integer = 8 + 2;
       →LetNode( Identifier(X), ExprNode( PlusNode( IntegerLiteral(8), IntegerLiteral(2) )))
```

[Marks: 10%]

Task 3 - Semantic Analysis, Execution and Interactivity

In this task you are to use the visitor design pattern (or any method you deem suitable) to traverse the parse tree to perform type-checking and execute the parse tree nodes. Remember that it is essential to have a proper implementation of a symbol table which is used by both stages.

The language is designed in such a way that one statement is a valid program in itself. This fact makes it easier for the interpreter to run in an interactive mode. This can be achieved by creating a main class called say SFLI (Simple Functional Language Interactive) which acts as an interactive console class. SFLI is to wrap an instance of the parser and an instance of the symbol table.

When SFLI is started, a prompt is presented to the user for him/her to type in statements to be executed. Once the statement is input, SFLI is required to run the parser, the type-checker and the interpreter respectively and output the result of the computation. It is convenient that the interpreter maintains a special variable (in functional languages this is usually called “it” or “ans”) which holds the last result computed. Below is an example of an interactive session.

`Sfl> let x : int = 8 + 2;` `Val ans : int = 10`	Creates a variable ‘x’ and assigns 10 to it
`Sfl> 24 + 12;` `Val ans : int = 36`	Computes the expression and stores it in ‘ans’
`Sfl> let y = ans * 2;` `Val ans : int = 72`	Creates a variable ‘x’ and assigns result to it
`Sfl> ans * 1.5;` `Type mismatch:  integer and real!`	Error since the types do not match

You can add an SFLI command to load scripts e.g.

```
Sfl> #load “factorial.sfl”
```

Note that for any direct SFLI commands a parser is not require, a simple string comparison is enough. One can enhance SFLI with a number of commands/functions for example; **#load** (to load scripts), **#quit** (to end the session), **#st** (displays the contents of the symbol-table) … This will aid in the debugging of you parser, type-checker and interpreter.

[Marks: 20%]

Task 4 - Sample programs

Together with the above, you are to design and implement short sample source programs to test the outcome of your compiler. In your report, state what you are testing for, insert the programs' parse tree and the outcome of your test.

[Marks: 20%]

The Report

In addition to the source and class files, you are to deliver a report. In your report include any deviations from the original EBNF, the salient points on how you developed the parser / interpreter (and reasons behind any decisions you took) including semantic rules and code execution, and any sample SFL programs you developed for testing.

[Marks: 25%]

Final Notes

As an example, the SFL source script below, computes the answer of a real number raised to an integer power:

```
// power function
function pow( x : real, n : int ) : real
{
    let y : real = 1.0;       //Declare y and set it to 1.0
    if( n>0 )
    {
        while(n>0)
        {
            y <- y * x;       //Assignment y = y * x;
            n <- n - 1;       //Assignment n = n - 1;
        }
    }
    else
    {
        while(n<0)
        {
            y <- y / x;       //Assignment y = y / x;
            n <- n + 1;       //Assignment n = n + 1;
        }
    }
    y;                        //return y as the result
}
```

Assuming that the above function is defined in a script file called "`power.sfl`", in SFLI we can make use of the function as follows:

```
Sfl>#load power.sfl

Sfl>pow(3.0, 2);
Ans : Real = 9.0

Sfl>
```